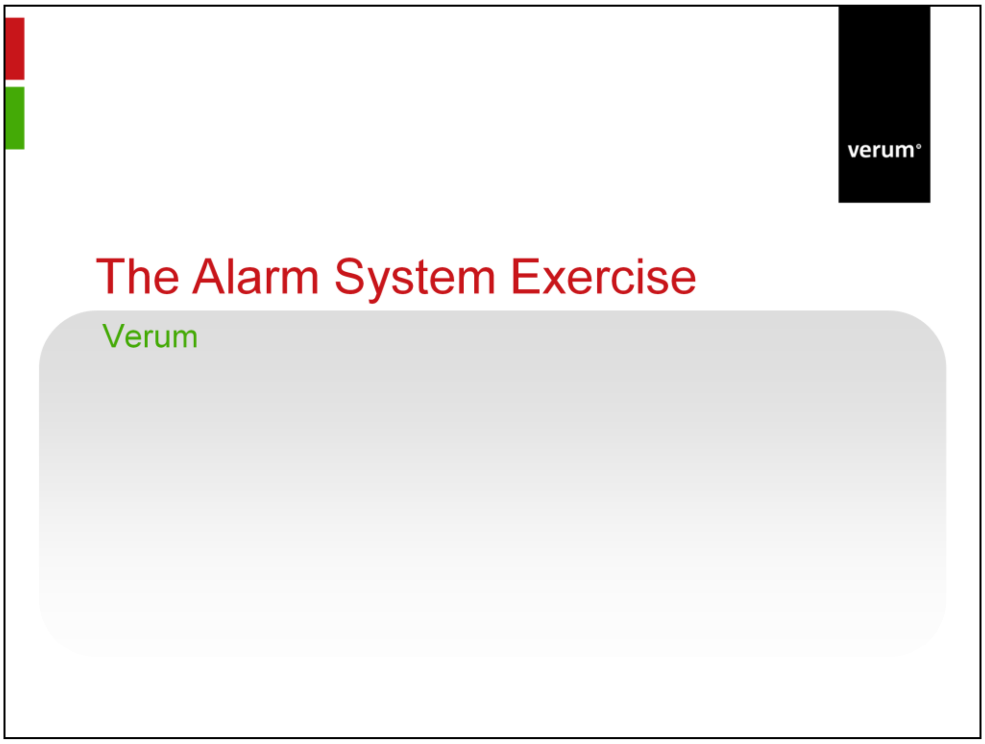
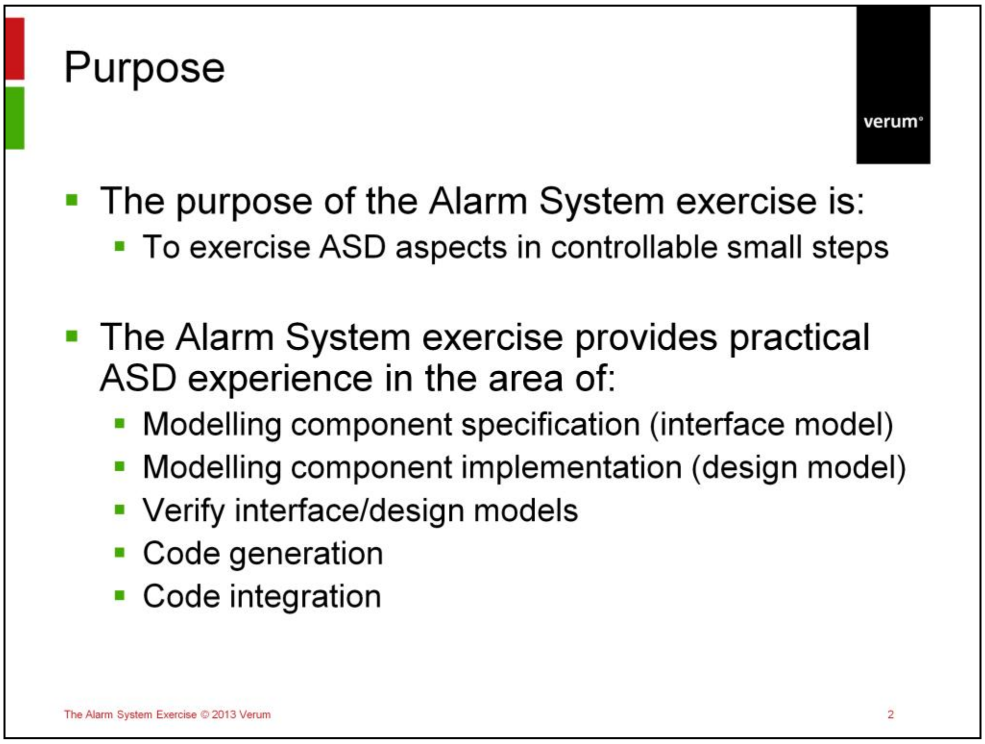


verum°

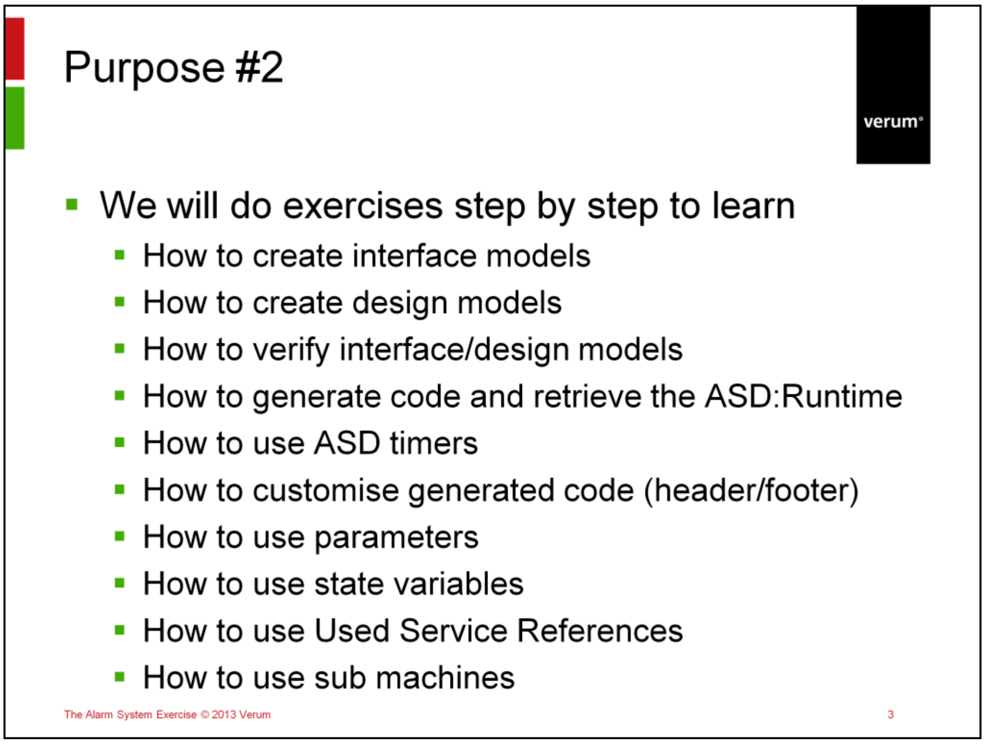
The Alarm System Exercise

Verum


Purpose

- The purpose of the Alarm System exercise is:
  - To exercise ASD aspects in controllable small steps
- The Alarm System exercise provides practical ASD experience in the area of:
  - Modelling component specification (interface model)
  - Modelling component implementation (design model)
  - Verify interface/design models
  - Code generation
  - Code integration

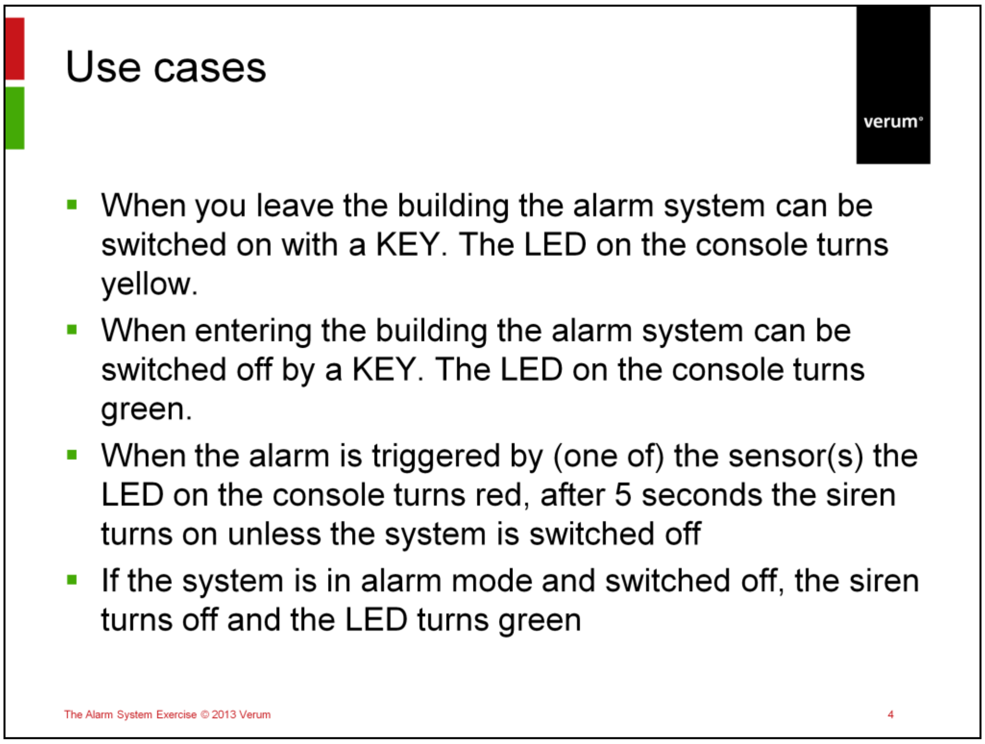
The Alarm System Exercise © 2013 Verum 2



Purpose #2

- We will do exercises step by step to learn
  - How to create interface models
  - How to create design models
  - How to verify interface/design models
  - How to generate code and retrieve the ASD:Runtime
  - How to use ASD timers
  - How to customise generated code (header/footer)
  - How to use parameters
  - How to use state variables
  - How to use Used Service References
  - How to use sub machines

The Alarm System Exercise © 2013 Verum 3



Use cases

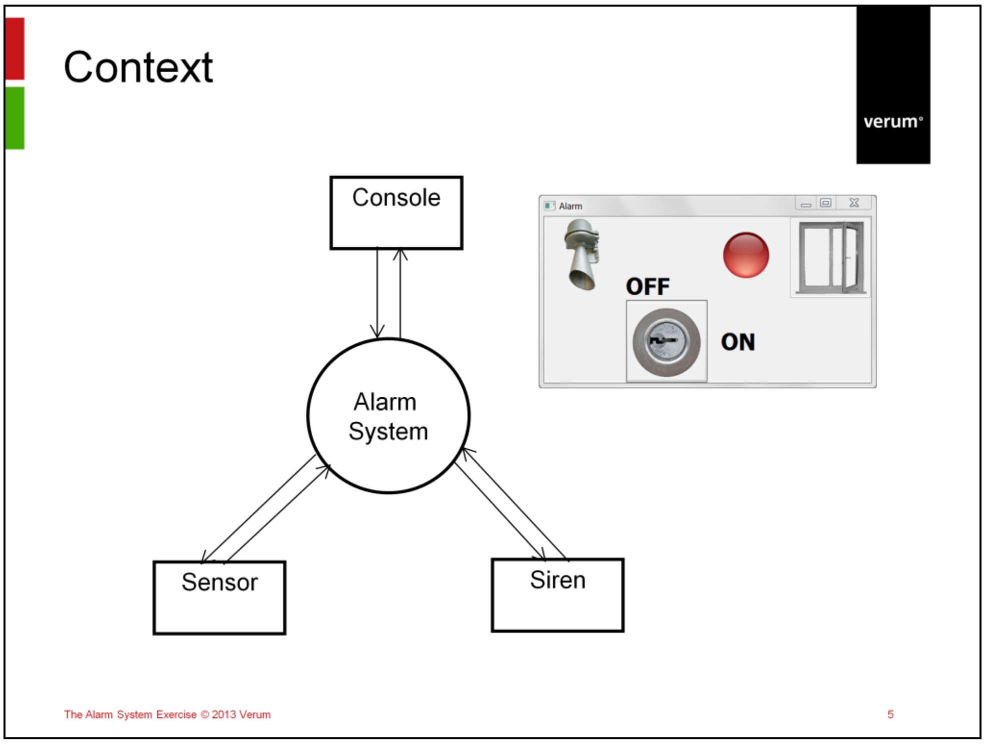
verum

- When you leave the building the alarm system can be switched on with a KEY. The LED on the console turns yellow.
- When entering the building the alarm system can be switched off by a KEY. The LED on the console turns green.
- When the alarm is triggered by (one of) the sensor(s) the LED on the console turns red, after 5 seconds the siren turns on unless the system is switched off
- If the system is in alarm mode and switched off, the siren turns off and the LED turns green

The Alarm System Exercise © 2013 Verum 4

These use cases describe the alarm system when it is complete. The exercises will be done step by step (incremental development) so the intermediate result may differ from these use cases. For example the first version of your alarm system will not have a time delay between the moment the sensor is triggered and the moment the siren is switched on.

At the Console switching off the siren must be done by switching off the entire system.



Context

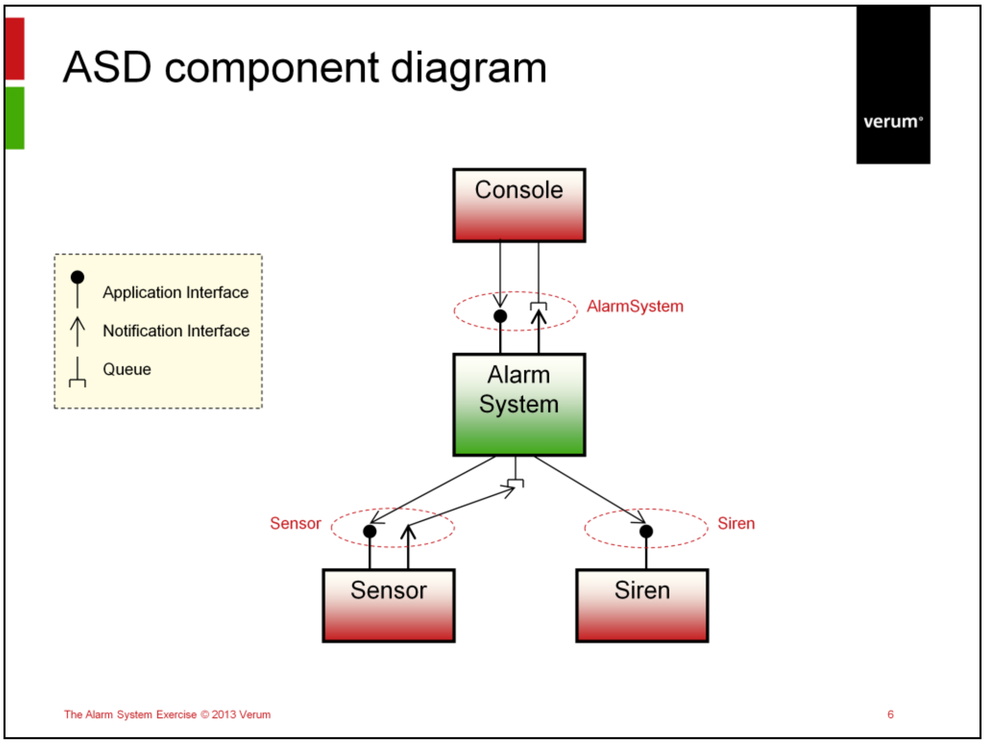
verum

The Alarm System Exercise © 2013 Verum 5

The Console is the Man Machine Interface (MMI) of the AlarmSystem. It consists of a key switch to switch the system on or off and a LED to indicate the status (green=OFF, yellow = ON, red = TRIGGERED - being pre-alarm or alarm).

The graphical representation of Siren and Sensor as shown on the Console GUI are actually not part of the Console itself but are shown for validation testing.

The Console will be implemented by a GUI application which is ready to use to test the result of your exercises.



ASD component diagram

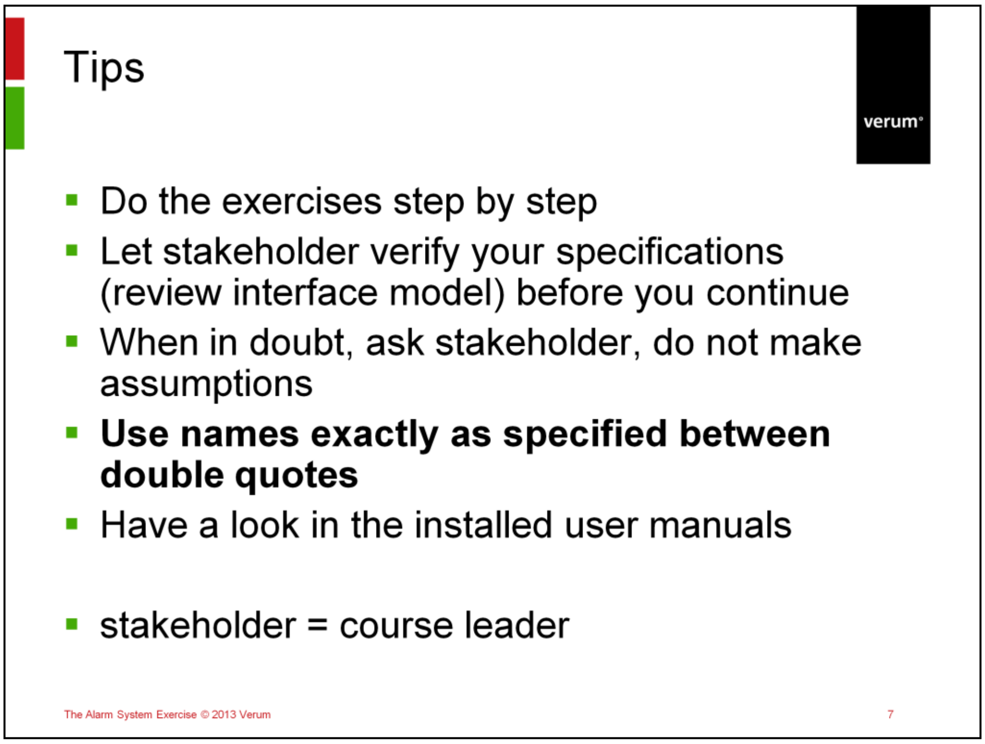
The Alarm System Exercise © 2013 Verum 6

The service of the Sensor and Siren is described in their respective ASD Interface Model.

The implementation of Console, Sensor and Siren can be found in the corresponding handwritten components.

You will make the Interface Model and Design Model of the AlarmSystem during the exercises.

Save your models in <desktop>\AlarmSystemExercise\models!



Tips

verum®

- Do the exercises step by step
- Let stakeholder verify your specifications (review interface model) before you continue
- When in doubt, ask stakeholder, do not make assumptions
- **Use names exactly as specified between double quotes**
- Have a look in the installed user manuals

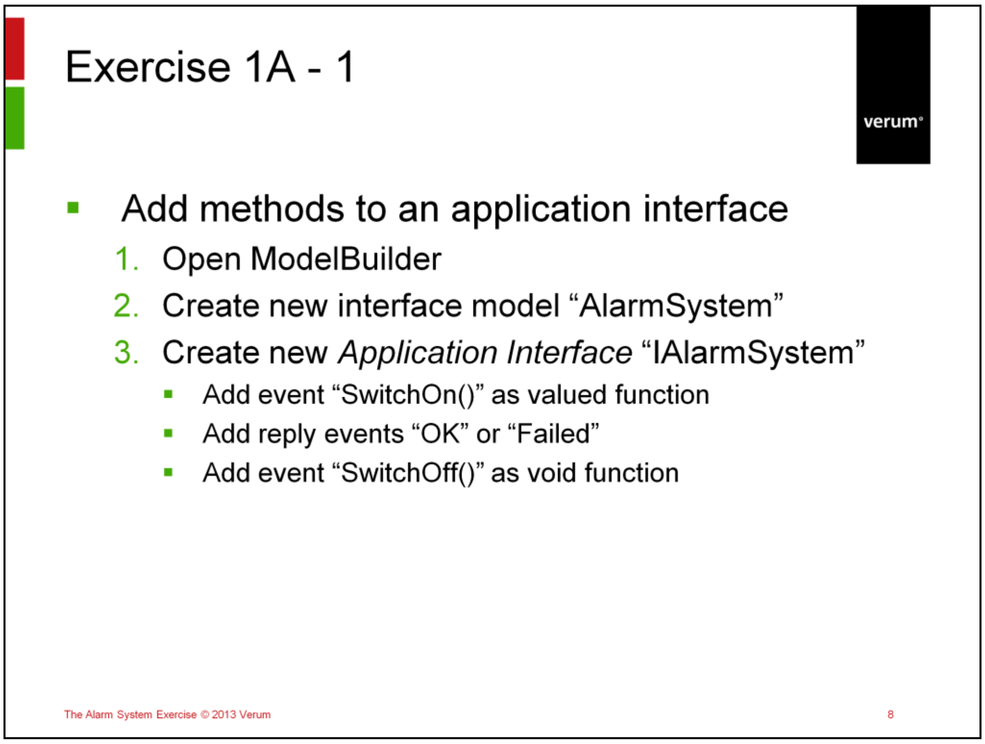
- stakeholder = course leader

The Alarm System Exercise © 2013 Verum 7

Not only in this course but also in a real project you should involve the stakeholders to validate that you are making the correct product. Making temporary assumptions is OK but add the fact you made an assumption as comment or tag and let the interface models be reviewed by the stakeholder(s).

User guides are available online (http://community.verum.com/documentation.aspx). Use them!

For more background information please consult the ASD community (http://community.verum.com).


Exercise 1A - 1

- Add methods to an application interface
  1. Open ModelBuilder
  2. Create new interface model “AlarmSystem”
  3. Create new *Application Interface* “IAlarmSystem”
     - Add event “SwitchOn()” as valued function
     - Add reply events “OK” or “Failed”
     - Add event “SwitchOff()” as void function

The Alarm System Exercise © 2013 Verum 8

In this exercise you learn to add the functionality of switching on and off the alarm system (using a key) to an ASD interface model.

Learning targets:

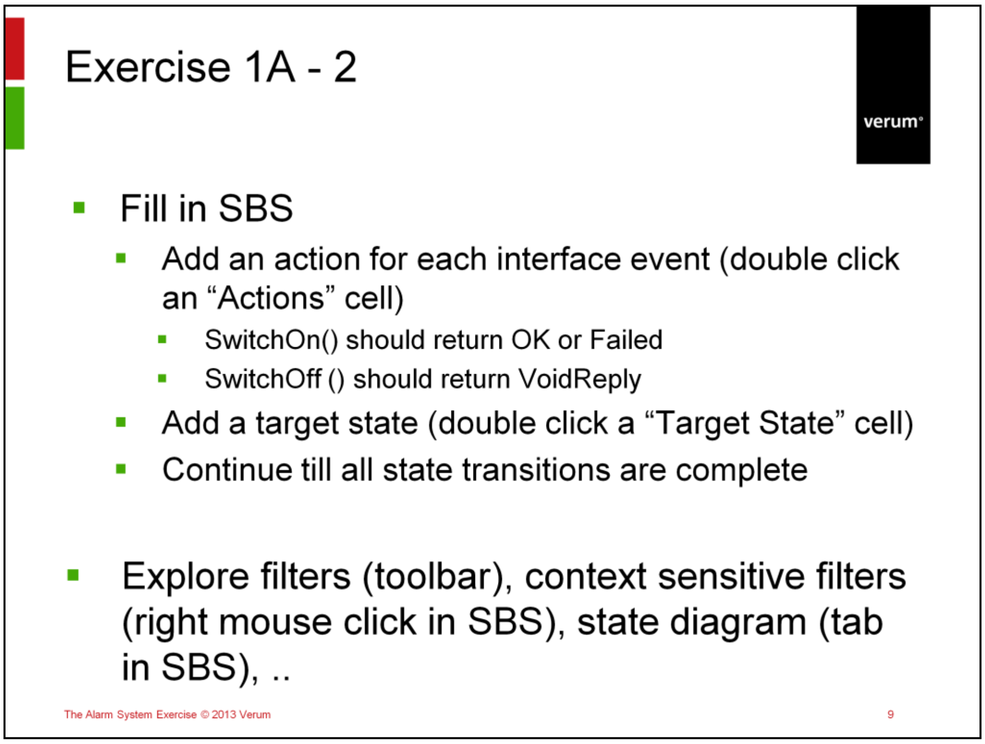
- How to create an interface model from scratch
- How to define application interfaces and events
- Non determinism in the interface
- How to use Sequence Based Specification (SBS)
- How to use the ASD:ModelBuilder (Conflict checker, Filters, Show State diagram, Navigation, ..)

References course material:

- Section 4. Sequence Based Specifications - Key concepts and best practices
- Section 6. ASD Interface Models - Sequence Based Specifications Guidelines

References user manual(s):

- ASD:Suite User Manual



Exercise 1A - 2

verum

- Fill in SBS
  - Add an action for each interface event (double click an "Actions" cell)
    - SwitchOn() should return OK or Failed
    - SwitchOff () should return VoidReply
  - Add a target state (double click a "Target State" cell)
  - Continue till all state transitions are complete
- Explore filters (toolbar), context sensitive filters (right mouse click in SBS), state diagram (tab in SBS), ..

The Alarm System Exercise © 2013 Verum 9

In this exercise you learn to add the functionality of switching on and off the alarm system (using a key) to an ASD interface model.

Learning targets:

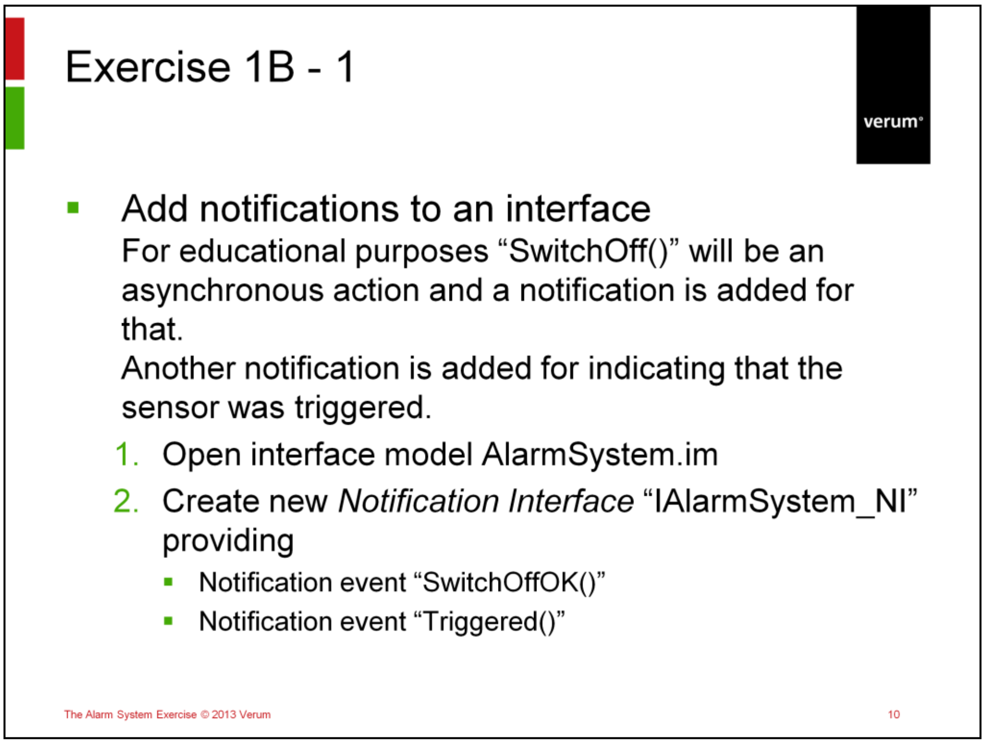
- How to create an interface model from scratch
- How to define application interfaces and events
- Non determinism in the interface
- How to use Sequence Based Specification (SBS)
- How to use the ASD:ModelBuilder (Rulecase wizards, Filters, Show State diagram, Navigation, ..)

References course material:

- Section 4. Sequence Based Specifications - Key concepts and best practices
- Section 6. ASD Interface Models - Sequence Based Specifications Guidelines

References user manual(s):

- ASD:Suite User Manual



Exercise 1B - 1

- **Add notifications to an interface**
  For educational purposes "SwitchOff()" will be an asynchronous action and a notification is added for that.
  Another notification is added for indicating that the sensor was triggered.
  1. Open interface model AlarmSystem.im
  2. Create new *Notification Interface* "IAlarmSystem_NI" providing
     - Notification event "SwitchOffOK()"
     - Notification event "Triggered()"

The Alarm System Exercise © 2013 Verum 10

In this exercise you learn to add notifications to an interface. The notifications sent to the client will depend on modelling events.

The Console will turn the LED green when the SwitchOffOK() notification is received.

The Console will turn the LED red when the Triggered() notification is received.

Learning targets:

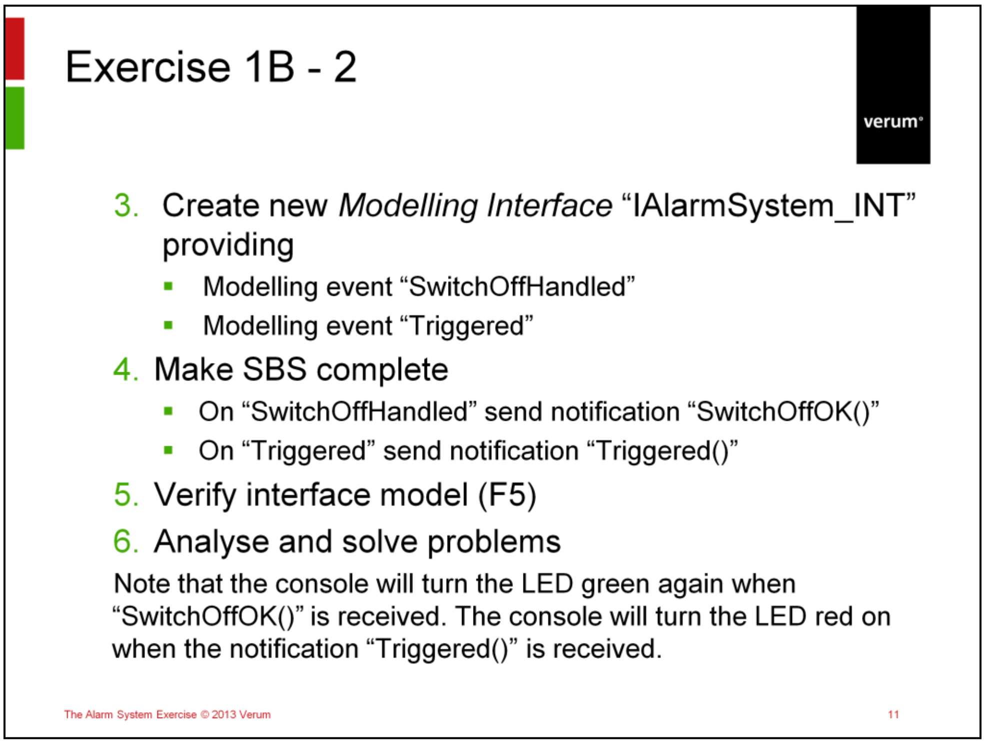
- How to define notification interfaces and notification events
- How to use Sequence Based Specification (SBS)
- How to use the ASD:ModelBuilder
- How to use the Visual Verification

References course material:

- Section 4. Sequence Based Specifications - Key concepts and best practices
- Section 6. ASD Interface Models - Sequence Based Specifications Guidelines
- Section 12. ASD:Suite Model verification - Definitions and best practices Model Verification

References user manual(s):

- ASD:Suite User Manual
- ASD:Suite Visual Verification Guide



Exercise 1B - 2

3. Create new *Modelling Interface* “IAlarmSystem_INT” providing
   - Modelling event “SwitchOffHandled”
   - Modelling event “Triggered”
4. Make SBS complete
   - On “SwitchOffHandled” send notification “SwitchOffOK()”
   - On “Triggered” send notification “Triggered()”
5. Verify interface model (F5)
6. Analyse and solve problems

Note that the console will turn the LED green again when “SwitchOffOK()” is received. The console will turn the LED red on when the notification “Triggered()” is received.

The Alarm System Exercise © 2013 Verum 11

In this exercise you learn to add a modelling interface. The modelling interface allows you to define internal situations that change the external observable behaviour (different state, different actions, ..).

When the AlarmSystem is requested to switch off you may want to wait till your entire system is really switched off. On the interface level you don’t need to know in detail what has been switched off but you only model the condition that the request to switch off has been handled i.e. “SwitchOffHandled”. The SwitchOffHandled condition may lead to a successful result or a failure. For this exercise we assume switching off has succeeded. The expected sequence is SwitchOff() – VoidReply – SwitchOffHandled – SwitchOffOK().

There can be a condition that will trip the alarm system, eventually leading to the siren being turned on. On the interface level you don’t need to model in detail what is the exact cause of the Triggered alarm (e.g. type of sensor) but you only model the effect that such a condition will have on the external state behaviour. In fact you make an abstraction i.e. “Triggered” of the internal behaviour.

Learning targets:

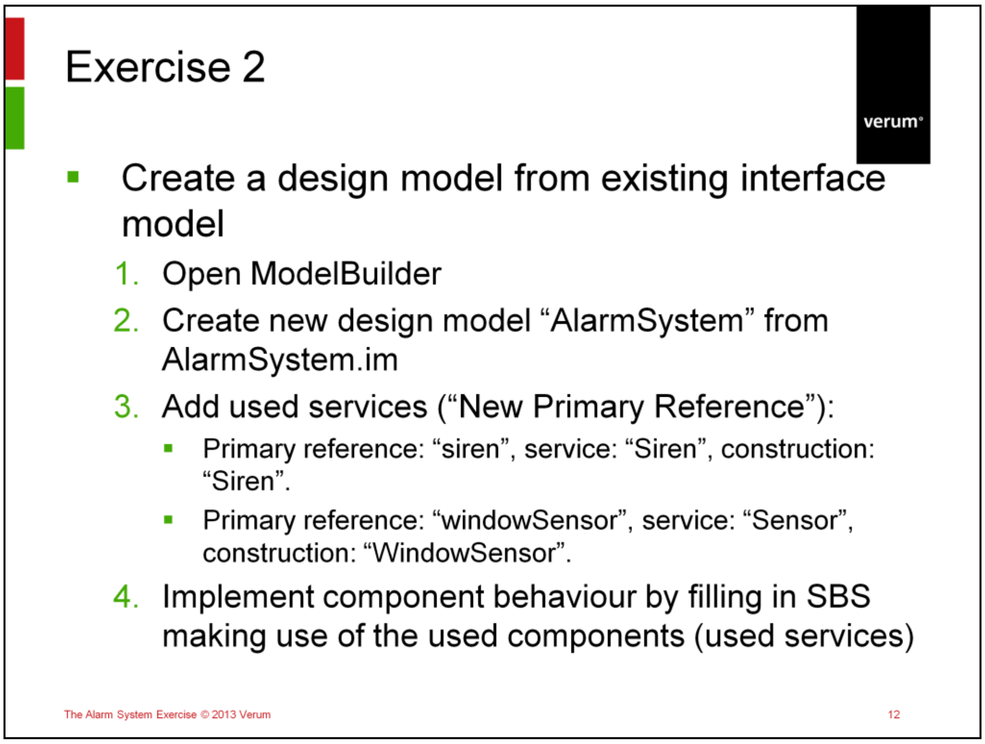
- How to define modelling events
- The difference between Optional and Inevitable
- How to use Sequence Based Specification (SBS)
- How to use the ASD:ModelBuilder and Visual Verification

References course material:

- Section 4. Sequence Based Specifications - Key concepts and best practices
- Section 6. ASD Interface Models - Sequence Based Specifications Guidelines
- Section 12. ASD:Suite Model verification - Definitions and best practices Visual Verification

References user manual(s):

- ASD:Suite User Manual
- ASD:Suite Visual Verification Guide


Exercise 2

verum®

- Create a design model from existing interface model
  1. Open ModelBuilder
  2. Create new design model “AlarmSystem” from AlarmSystem.im
  3. Add used services (“New Primary Reference”):
     - Primary reference: “siren”, service: “Siren”, construction: “Siren”.
     - Primary reference: “windowSensor”, service: “Sensor”, construction: “WindowSensor”.
  4. Implement component behaviour by filling in SBS making use of the used components (used services)

The Alarm System Exercise © 2013 Verum 12

In this exercise you learn to create an ASD design model from an already existing interface model.

In case the interface model has modelling events, the design model could initially have “floating states”. These floating states can be resolved by adding used components and use their output events as input events in the SBS. For example: the “Triggered” modelling event is equivalent to the notification event “SensorTriggered()” of the WindowSensor.

Note however that not all floating states have to be resolved; it can happen that the interface model is prepared for failure situations while the used components do not have these failure situations. In such cases a floating state can be removed.

Learning targets:

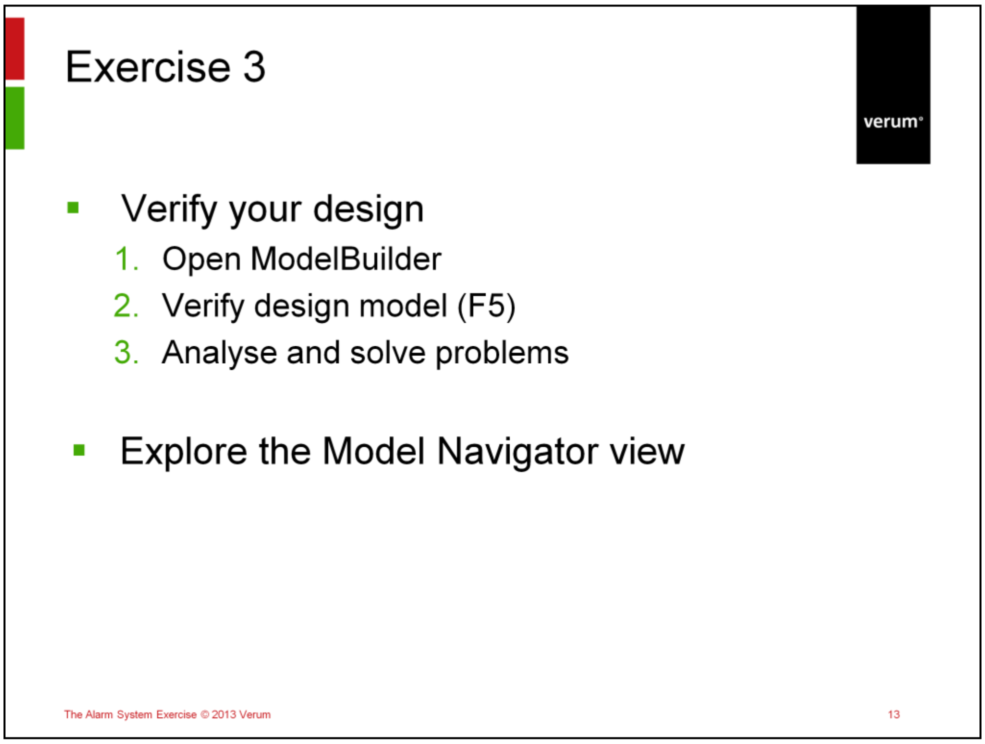
- How to create design model from existing interface model
- How to add used services
- How to fix floating states due to modelling events
- Use Sequence Based Specification (SBS)
- How to use the ASD:ModelBuilder

References course material:

- Section 4. Sequence Based Specifications - Key concepts and best practices
- Section 7. ASD Design Models - Sequence Based Specifications Guidelines

References user manual(s):

- ASD:Suite User Manual


Exercise 3

verum

- Verify your design
  1. Open ModelBuilder
  2. Verify design model (F5)
  3. Analyse and solve problems
- Explore the Model Navigator view

The Alarm System Exercise © 2013 Verum 13

In this exercise you learn to correct the defined behaviour by using Visual Verification and to analyse/solve the reported problems.

Learning targets:

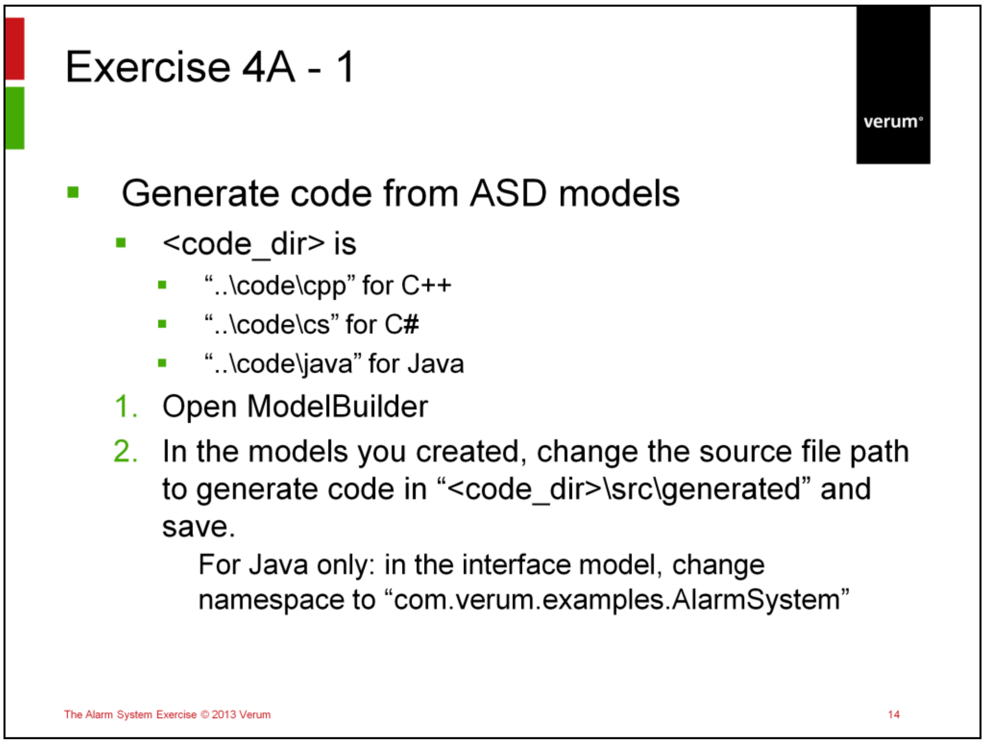
- How to use Visual Verification
- How to analyze design problems

References course material:

- Section 11. ASD - Execution Semantics
- Section 12. ASD:Suite Model verification - Definitions and best practices

References user manual(s):

- ASD:Suite Visual Verification Guide



Exercise 4A - 1

verum

- Generate code from ASD models
  - <code_dir> is
    - "..\code\cpp" for C++
    - "..\code\cs" for C#
    - "..\code\java" for Java
  1. Open ModelBuilder
  2. In the models you created, change the source file path to generate code in "<code_dir>\src\generated" and save.
     For Java only: in the interface model, change namespace to "com.verum.examples.AlarmSystem"

The Alarm System Exercise © 2013 Verum 14

In this exercise you learn to generate code from ASD models.

Each model has properties related to code generation; they can be filled in through the model properties. The chosen source file path must point to the location where the build environment expects the files i.e. <code_dir> (see slide).

Note that through the model properties also debug info can be included in the generated code.

Learning targets:

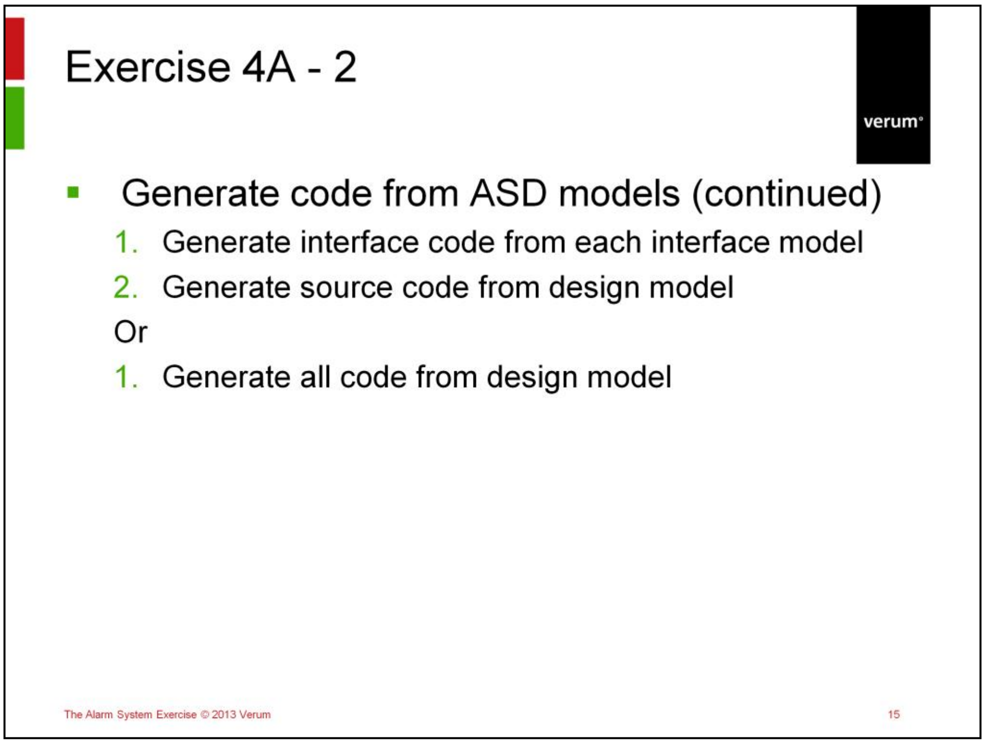
- How to use the Code Generator

References course material:

- Section 13. ASD:Suite - Code generation

References user manual(s):

- ASD:Runtime Guide


Exercise 4A - 2

- Generate code from ASD models (continued)
  1. Generate interface code from each interface model
  2. Generate source code from design model

  Or

  1. Generate all code from design model

The Alarm System Exercise © 2013 Verum 15

In this exercise you learn to generate code from ASD models.

Learning targets:

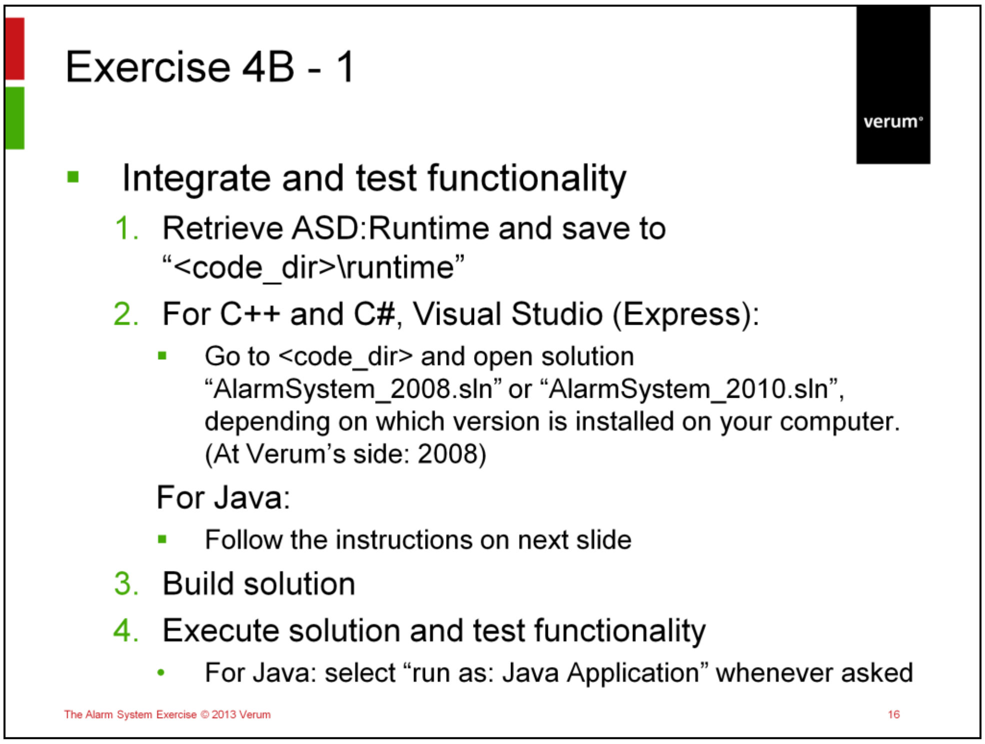
- How to use the Code Generator

References course material:

- Section 13. ASD:Suite - Code generation

References user manual(s):

• ASD:Runtime Guide



Exercise 4B - 1

- Integrate and test functionality
  1. Retrieve ASD:Runtime and save to “<code_dir>\runtime”
  2. For C++ and C#, Visual Studio (Express):
     - Go to <code_dir> and open solution “AlarmSystem_2008.sln” or “AlarmSystem_2010.sln”, depending on which version is installed on your computer. (At Verum’s side: 2008)

     For Java:
     - Follow the instructions on next slide
  3. Build solution
  4. Execute solution and test functionality
     - For Java: select “run as: Java Application” whenever asked

The Alarm System Exercise © 2013 Verum 16

In this exercise you learn to retrieve the ASD:Runtime and you experience the effort to integrate ASD runtime, ASD generated code and other code into a testable product.

Learning targets:

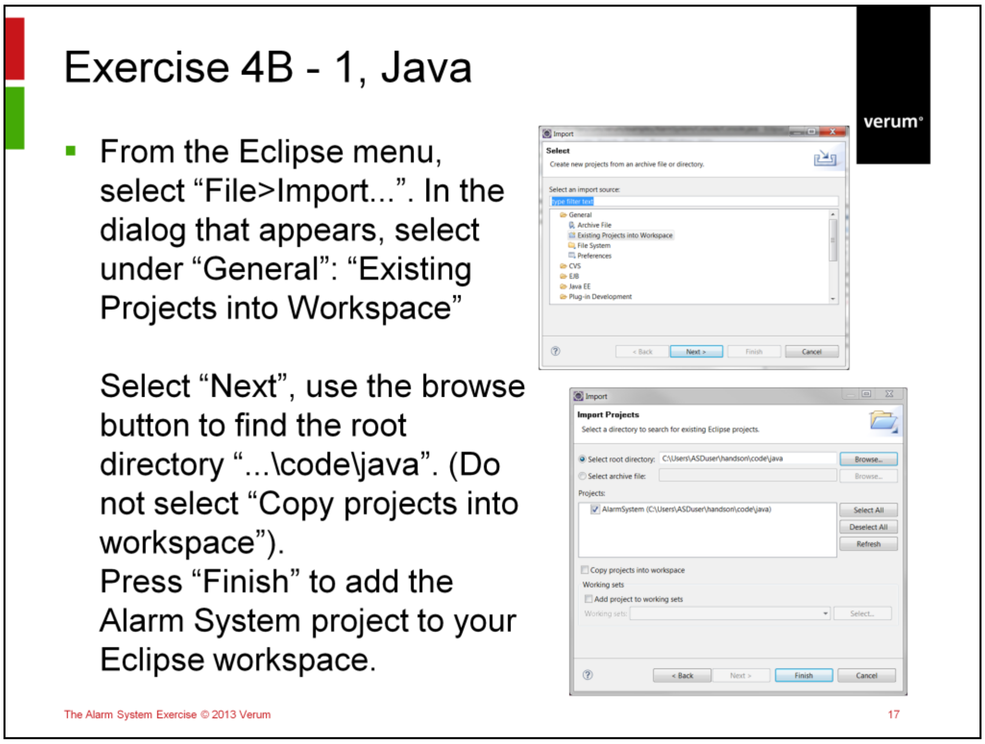
- How to download the ASD Runtime
- How to integrate ASD generated code with existing handwritten code
- Integration = easy job if interfaces are precise and complete

References course material:

- Section 13. ASD:Suite - Code generation
- Section 14. Software Integration - ASD:Runtime and OSAL

References user manual(s):

- ASD:Runtime Guide



Exercise 4B - 1, Java

- From the Eclipse menu, select “File>Import...”. In the dialog that appears, select under “General”: “Existing Projects into Workspace”

  Select “Next”, use the browse button to find the root directory “...\code\java”. (Do not select “Copy projects into workspace”).
  Press “Finish” to add the Alarm System project to your Eclipse workspace.

Import

Select

Create new projects from an archive file or directory.

Select an import source:

type filter text

- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Preferences
- CVS
- EJB
- Java EE
- Plug-in Development

Import

Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: C:\Users\ASDuser\handson\code\java

( ) Select archive file:

Projects:

[✓] AlarmSystem (C:\Users\ASDuser\handson\code\java)

[ ] Copy projects into workspace

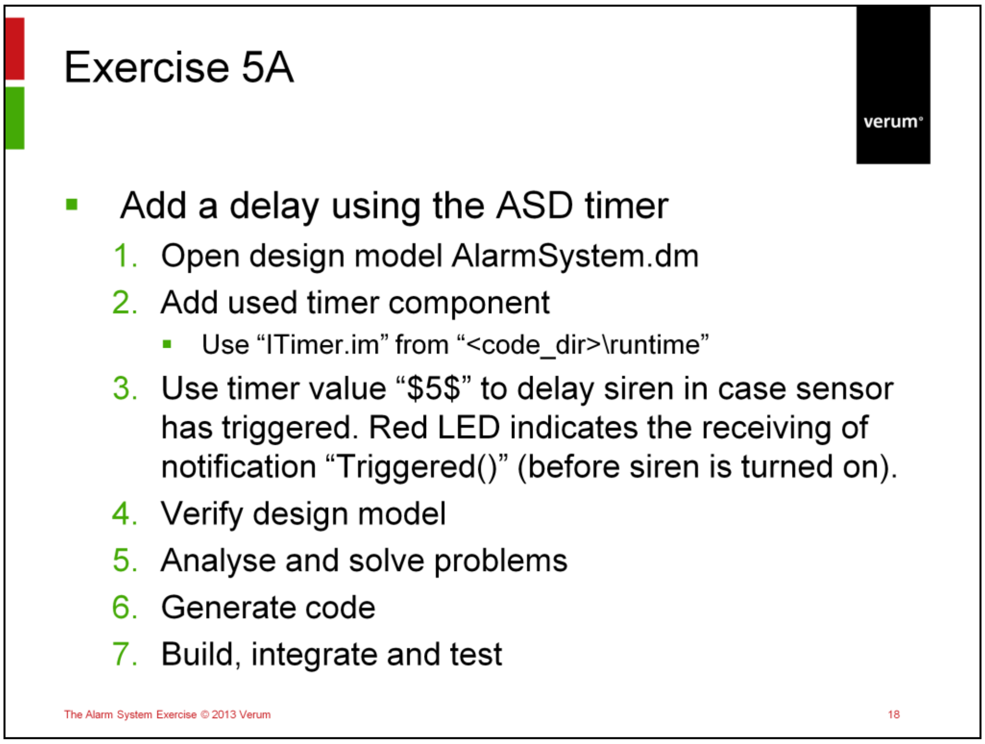
Working sets

[ ] Add project to working sets

Working sets:

The Alarm System Exercise © 2013 Verum 17

When ever you have generated new code using the ASD:ModelBuilder, perform a refresh in the Project Explorer of Eclipse to view the changes.



Exercise 5A

verum

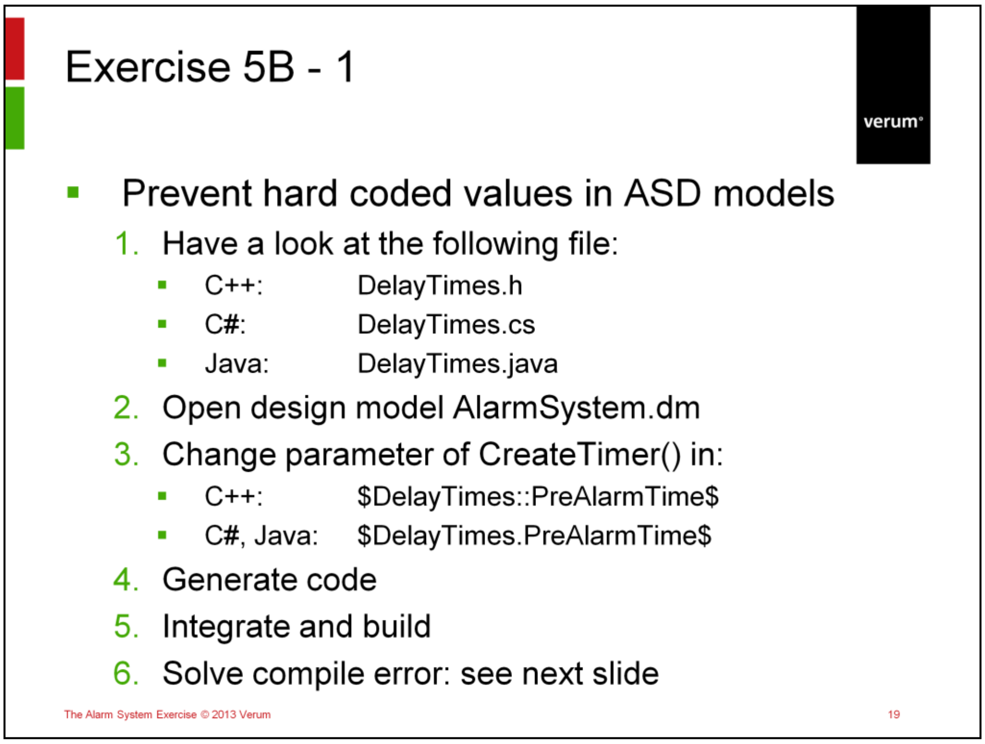
- Add a delay using the ASD timer
  1. Open design model AlarmSystem.dm
  2. Add used timer component
     - Use “ITimer.im” from “<code_dir>\runtime”
  3. Use timer value “5” to delay siren in case sensor has triggered. Red LED indicates the receiving of notification “Triggered()” (before siren is turned on).
  4. Verify design model
  5. Analyse and solve problems
  6. Generate code
  7. Build, integrate and test

The Alarm System Exercise © 2013 Verum 18

The Triggered() notification is the indication that there is an alarm condition (LED is turned red).

The delay between sending the notification Triggered() and the moment that the siren is actually switched on is to give you the opportunity to switch off the alarm system (if you have the key).

The value of the timer can be set by editing the CreateTimer(t) in the “Select Actions” editor. Change argument ”t” in “5”.



Exercise 5B - 1

- Prevent hard coded values in ASD models
  1. Have a look at the following file:

C++:	DelayTimes.h
C#:	DelayTimes.cs
Java:	DelayTimes.java

  2. Open design model AlarmSystem.dm
  3. Change parameter of CreateTimer() in:

C++:	$DelayTimes::PreAlarmTime$
C#, Java:	$DelayTimes.PreAlarmTime$

  4. Generate code
  5. Integrate and build
  6. Solve compile error: see next slide

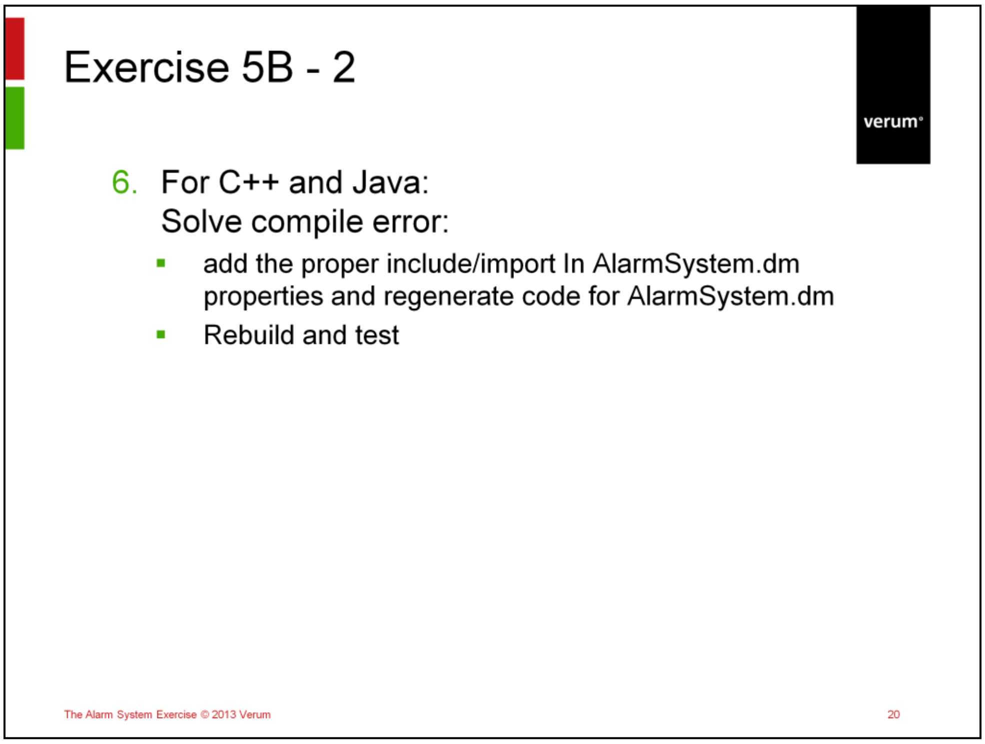
The Alarm System Exercise © 2013 Verum 19

In this exercise you learn how to prevent hard coded values as parameter in ASD models and instead decide upon this in the code (increases adaptability).

Learning targets:
- How use configurable literals as parameter

References course material:
- Section 10. ASD:Suite -Timers
- Section 13. ASD:Suite - Code generation
- Section 14. Software Integration - ASD:Runtime and OSAL

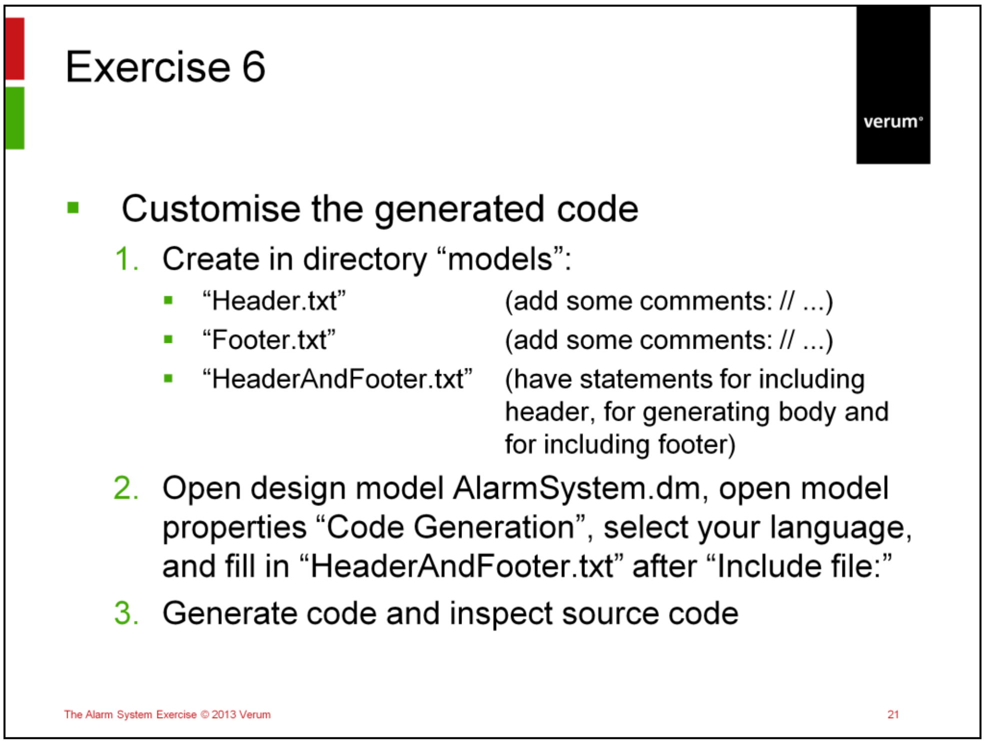
References user manual(s):
- ASD:Suite User Manual
- ASD:Runtime Guide



Exercise 5B - 2

6. For C++ and Java:
   Solve compile error:
   - add the proper include/import In AlarmSystem.dm properties and regenerate code for AlarmSystem.dm
   - Rebuild and test

The Alarm System Exercise © 2013 Verum 20


Exercise 6

verum

- Customise the generated code
  1. Create in directory "models":
     - "Header.txt" (add some comments: // ...)
     - "Footer.txt" (add some comments: // ...)
     - "HeaderAndFooter.txt" (have statements for including header, for generating body and for including footer)
  2. Open design model AlarmSystem.dm, open model properties "Code Generation", select your language, and fill in "HeaderAndFooter.txt" after "Include file:"
  3. Generate code and inspect source code

The Alarm System Exercise © 2013 Verum 21

In this exercise you learn how to customise the generated code.

Learning targets:

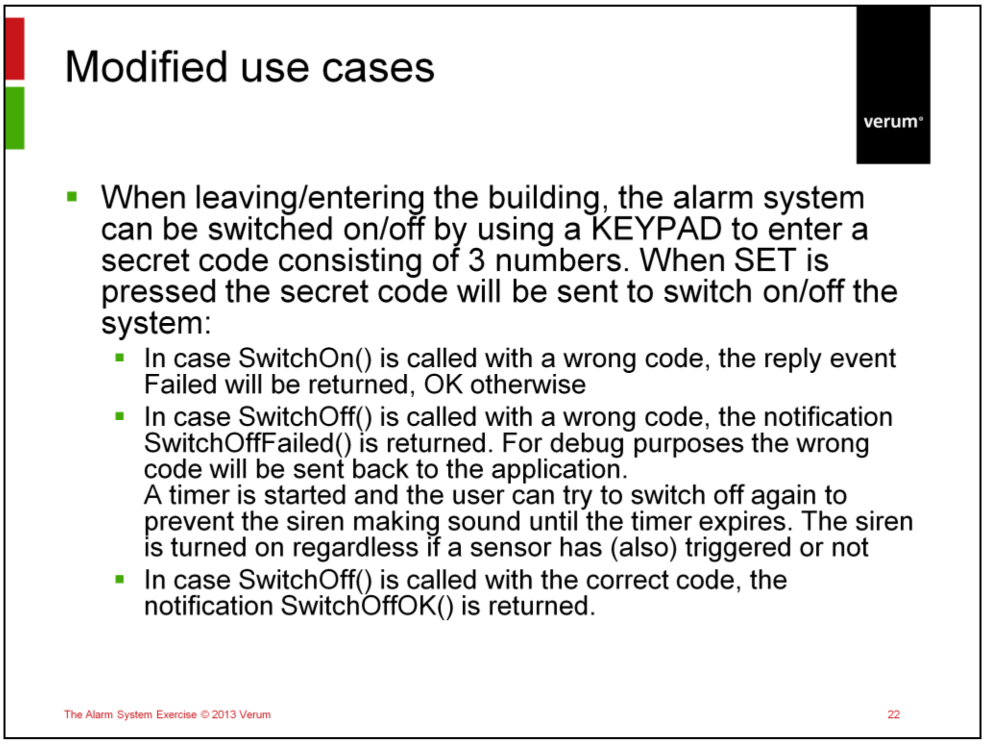
- How to customize generated code

References course material:

- Section 13. ASD:Suite - Code generation

References user manual(s):

- ASD:Suite User Manual
  (keyword "Code Customization", User Provided Text)
- ASD:Runtime Guide



Modified use cases

verum

- When leaving/entering the building, the alarm system can be switched on/off by using a KEYPAD to enter a secret code consisting of 3 numbers. When SET is pressed the secret code will be sent to switch on/off the system:
  - In case SwitchOn() is called with a wrong code, the reply event Failed will be returned, OK otherwise
  - In case SwitchOff() is called with a wrong code, the notification SwitchOffFailed() is returned. For debug purposes the wrong code will be sent back to the application.
    A timer is started and the user can try to switch off again to prevent the siren making sound until the timer expires. The siren is turned on regardless if a sensor has (also) triggered or not
  - In case SwitchOff() is called with the correct code, the notification SwitchOffOK() is returned.

The Alarm System Exercise © 2013 Verum 22

Our first specification of the AlarmSystem is extended to allow exercising on more advanced features.

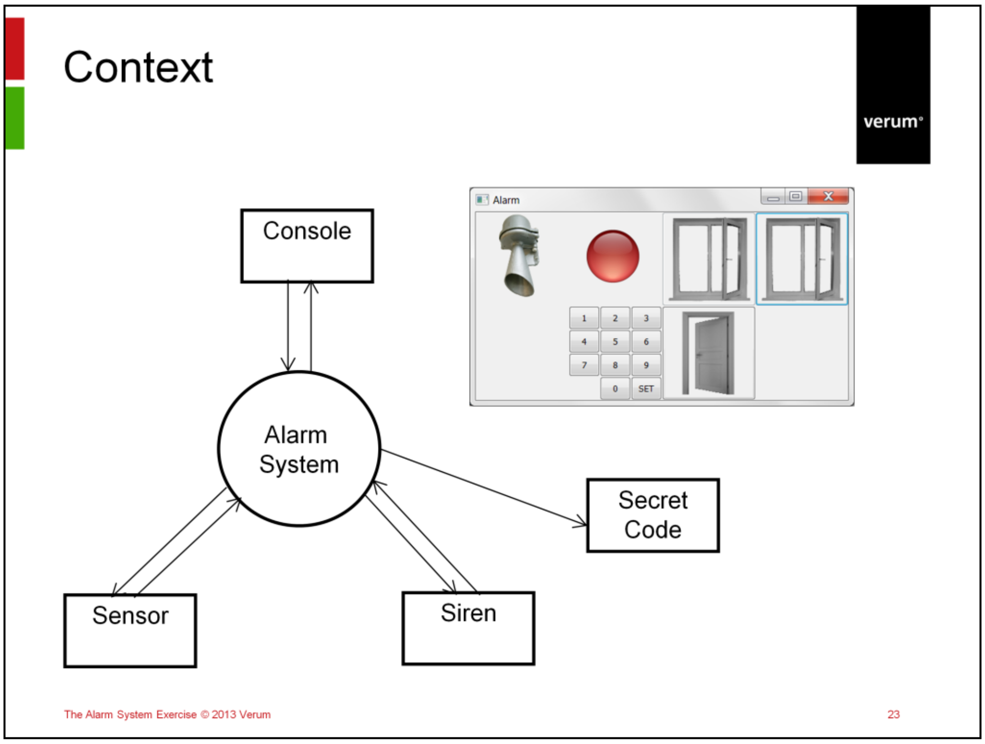
We already had the sequence SwitchOff() – VoidReply – SwitchOffHandled – SwitchedOffOK() from the previous exercises.

We now add the sequence SwitchOff() – VoidReply – SwitchOffHandled – SwitchedOffFailed()

These sequences are mutually exclusive.

In case of a wrong code the console will turn the LED red when notification SwitchOffFailed() is received. The state of the alarm system in that case is the same as if a sensor was triggered. This means the siren will make sound after a delay unless the correct code is entered in the mean time. In this phase the red LED will indicate the pre-alarm.

The secret code is “917” (hard-coded in the implementation of the handwritten component SecretCode)



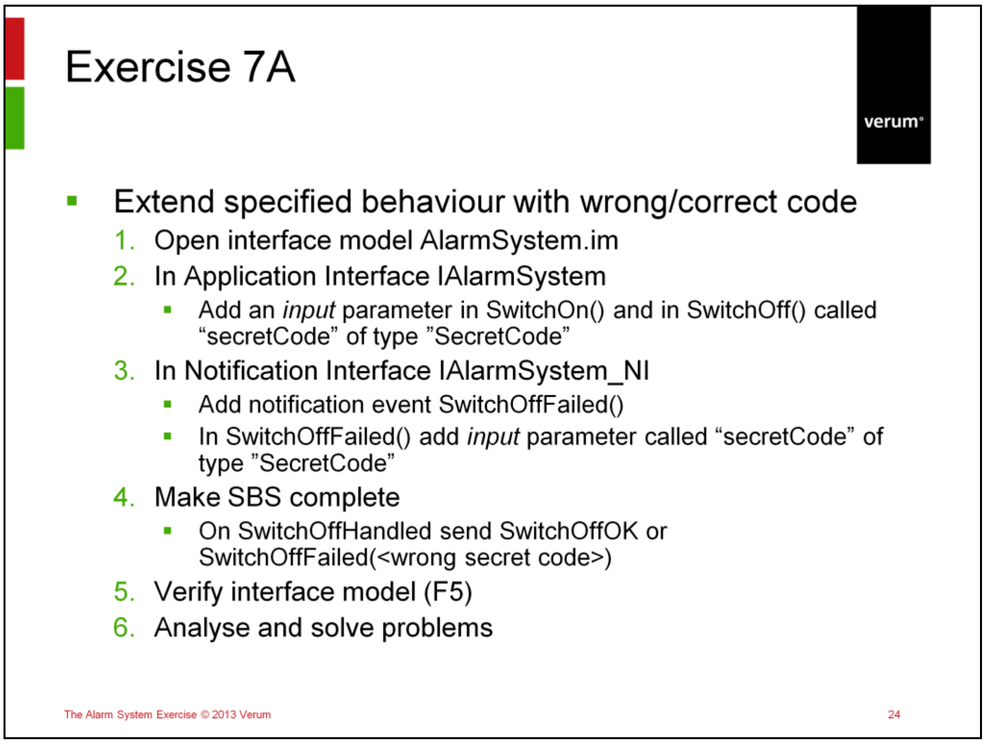
Context

The interface of the SecretCode, Sensor and Siren are described in their respective Interface Models.

The implementation of Console, Sensor and Siren can be found in the corresponding handwritten components.

You will make the Interface Model and Design Model of the AlarmSystem during the exercises.

Save your models in <desktop>\AlarmSystemExercise\models!



Exercise 7A

- Extend specified behaviour with wrong/correct code
  1. Open interface model AlarmSystem.im
  2. In Application Interface IAlarmSystem
     - Add an *input* parameter in SwitchOn() and in SwitchOff() called “secretCode” of type ”SecretCode”
  3. In Notification Interface IAlarmSystem_NI
     - Add notification event SwitchOffFailed()
     - In SwitchOffFailed() add *input* parameter called “secretCode” of type ”SecretCode”
  4. Make SBS complete
     - On SwitchOffHandled send SwitchOffOK or SwitchOffFailed(<wrong secret code>)
  5. Verify interface model (F5)
  6. Analyse and solve problems

The Alarm System Exercise © 2013 Verum 24

In this exercise you learn how to define data exchange as input and output parameter.

The behaviour is modified to handle the wrong or correct code according the modified use cases.

If the code is correct the system will be switched off (giving the response “SwitchOffOK()”).

If the system is not correct the system will be triggered which is indicated by “SwitchOffHandled” giving the response “SwitchOffFailed()”.

After 10 seconds the siren will make sound unless a correct code is entered in the mean time. If another incorrect code is entered the timer will not be restarted.

If the debug option is switched on in the generated code you will be able to read in the logging window which wrong code has been used to switch off the system.

Learning targets:

- How to define parameters in application and notification events in an interface model

Note that ASD can be used in an iterative or a waterfall development process.

The exercise is described as a waterfall approach. It is however possible to make iterations in exercise 7.

For instance, first extend the model to switch on the system with a correct secret code. Check models, analyse/solve problems, generate code and integrate and test.

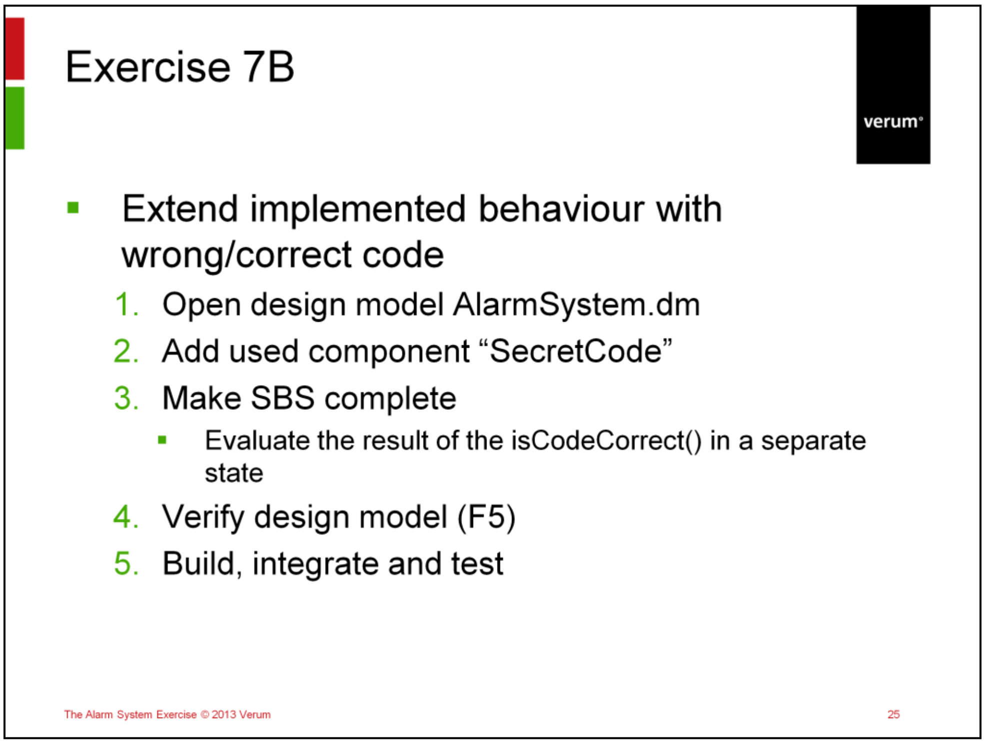
If it works OK with the GUI, then start modelling switch off functionality.

References course material:

- Section 8. Parameters - Key concepts and best practices

References user manual(s):

• ASD:Suite User Manual



Exercise 7B

verum

- Extend implemented behaviour with wrong/correct code
  1. Open design model AlarmSystem.dm
  2. Add used component “SecretCode”
  3. Make SBS complete
     - Evaluate the result of the isCodeCorrect() in a separate state
  4. Verify design model (F5)
  5. Build, integrate and test

The Alarm System Exercise © 2013 Verum 25

In this exercise you learn how to use input and output data in the design model.

Learning targets:

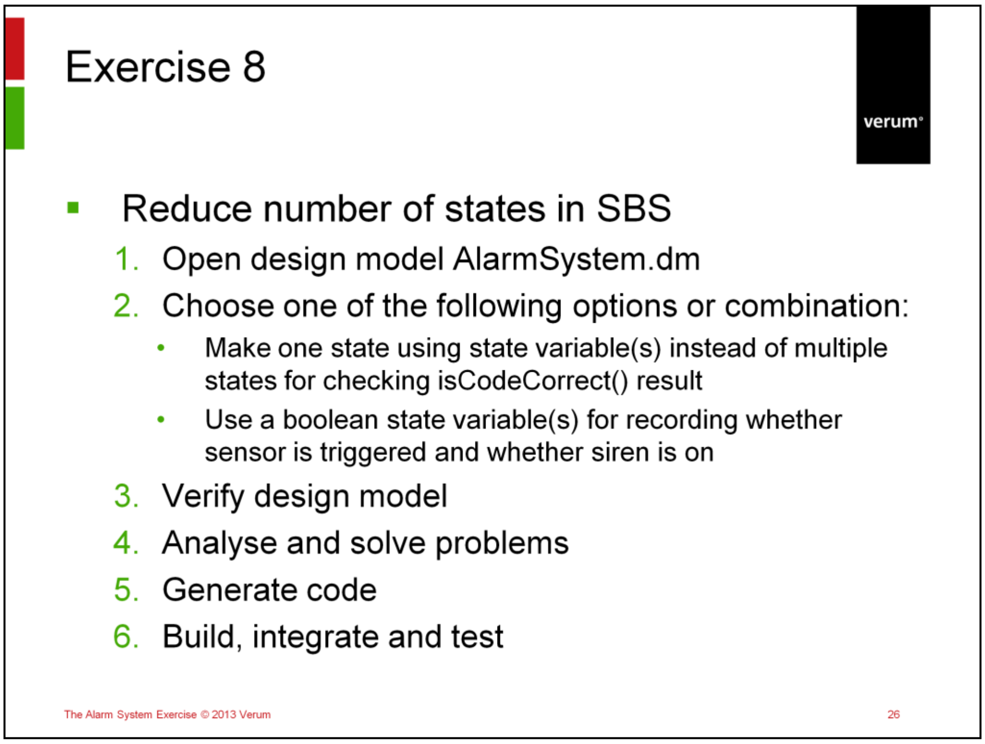
- How to use parameters in application and notification events in a design model
- How to use storage specifiers in design models
- How to implement data types defined/used in ASD models
- ASD does not guarantee data integrity (experienced if storage specifiers are not used well)
- Conversion of data parameters into ASD control (isCodeCorrect() function)
- Note the alike behaviour in case of checking result of isCodeCorrect(), multiple states looking the same except a small difference

References course material:

- Section 8. Parameters - Key concepts and best practices
- Section 15. Software Integration - Foreign components
- Section 16. Using Parameters - Generated code

References user manual(s):

- ASD;Suite User Manual



Exercise 8

verum

- Reduce number of states in SBS
  1. Open design model AlarmSystem.dm
  2. Choose one of the following options or combination:
     - Make one state using state variable(s) instead of multiple states for checking isCodeCorrect() result
     - Use a boolean state variable(s) for recording whether sensor is triggered and whether siren is on
  3. Verify design model
  4. Analyse and solve problems
  5. Generate code
  6. Build, integrate and test

The Alarm System Exercise © 2013 Verum 26

In this exercise you learn how you can condense a number of states that almost do the same to one state by using state variables.

By reducing the number of states the human perceived complexity in the SBS may decrease. The model verification however will use the exact same state space as without state variables.

Learning targets:

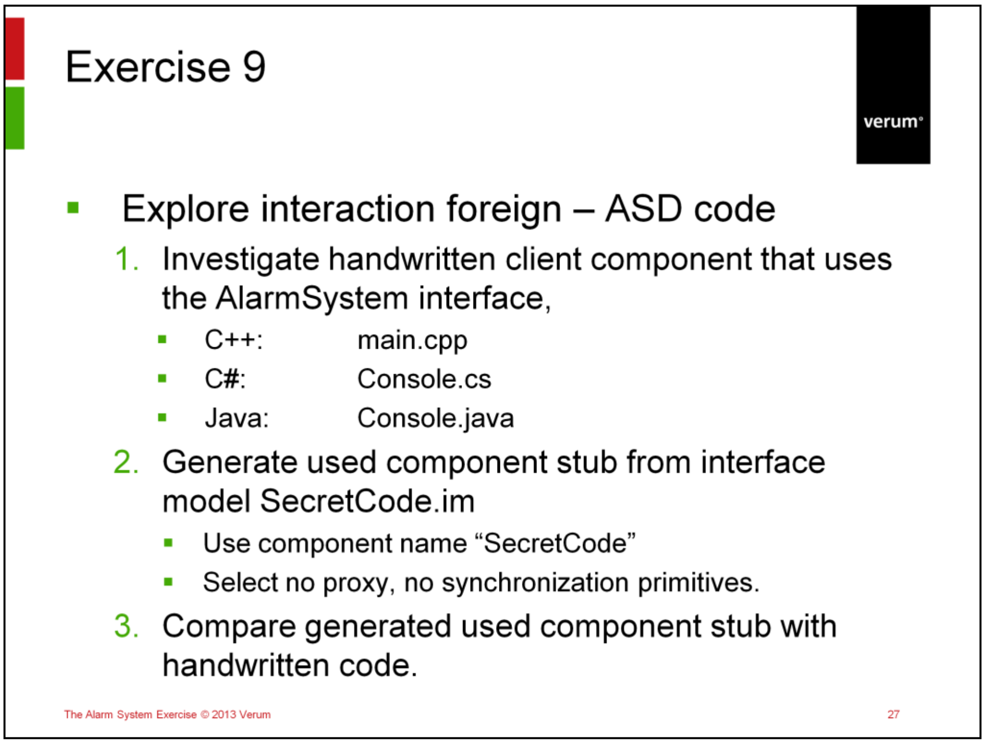
- How to use state variables to condense SBS states

References course material:

- Section 5. State Variables - Key concepts and best practices

References user manual(s):

- ASD:Suite User Manual



Exercise 9

verum®

- Explore interaction foreign – ASD code
  1. Investigate handwritten client component that uses the AlarmSystem interface,

C++:	main.cpp
C#:	Console.cs
Java:	Console.java

  2. Generate used component stub from interface model SecretCode.im
     - Use component name “SecretCode”
     - Select no proxy, no synchronization primitives.
  3. Compare generated used component stub with handwritten code.

The Alarm System Exercise © 2013 Verum 27

In this exercise you can have a look how:

- A foreign client component interacts with the ASD component i.e. AlarmSystem
- How an ASD component interacts with a foreign used component. This component can initially be generated from an ASD Model Interface and must be extended manually to fulfil its extended functionality.

Learning targets:

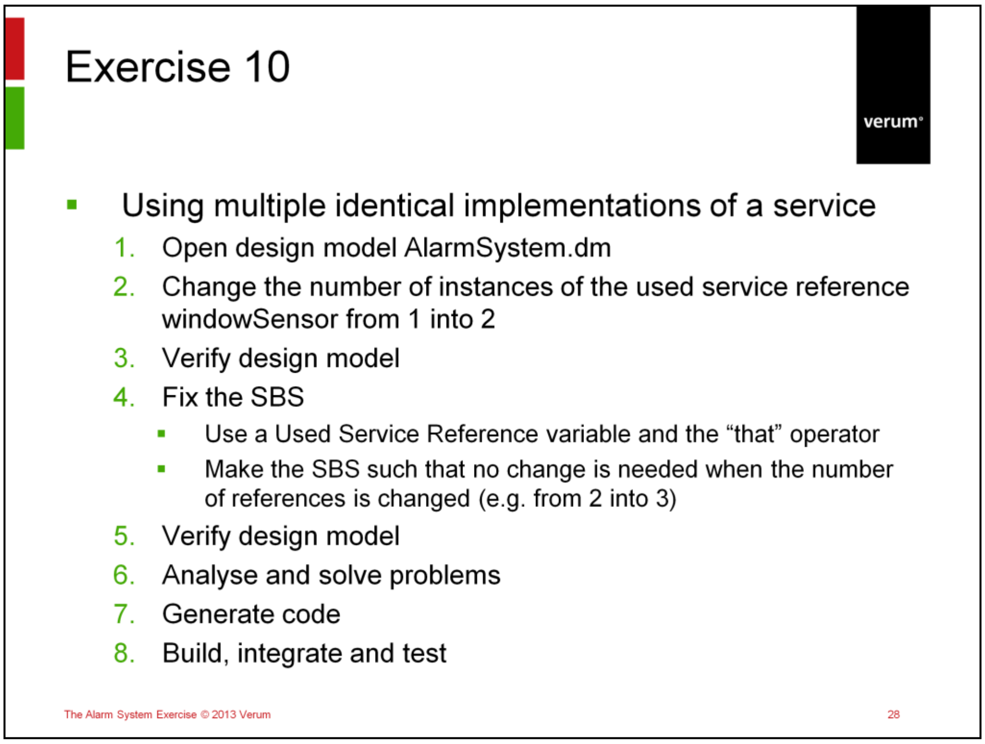
- How to write software that interacts with ASD components (used component, client component)

References course material:

- Section 15. Software Integration - Foreign components

References user manual(s):

- ASD:Runtime Guide



Exercise 10

verum®

- Using multiple identical implementations of a service
  1. Open design model AlarmSystem.dm
  2. Change the number of instances of the used service reference windowSensor from 1 into 2
  3. Verify design model
  4. Fix the SBS
     - Use a Used Service Reference variable and the “that” operator
     - Make the SBS such that no change is needed when the number of references is changed (e.g. from 2 into 3)
  5. Verify design model
  6. Analyse and solve problems
  7. Generate code
  8. Build, integrate and test

The Alarm System Exercise © 2013 Verum 28

In this exercise you learn how you can create a set of identical implementations of the same interface and how you can use Used Service References to address a set of components with the same interface.

Learning targets:

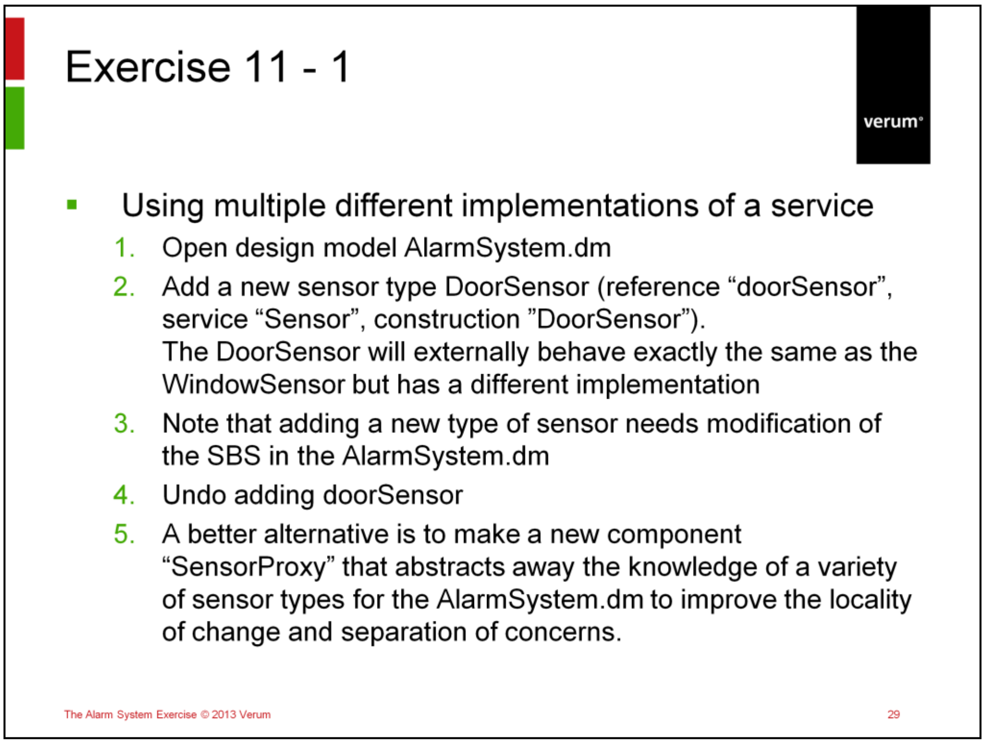
- How to use used service references
- How to use used service reference state variables and “that” operator

References course material:

- -

References user manual(s):

- ASD:Suite User Manual


Exercise 11 - 1

- Using multiple different implementations of a service
  1. Open design model AlarmSystem.dm
  2. Add a new sensor type DoorSensor (reference “doorSensor”, service “Sensor”, construction ”DoorSensor”).
     The DoorSensor will externally behave exactly the same as the WindowSensor but has a different implementation
  3. Note that adding a new type of sensor needs modification of the SBS in the AlarmSystem.dm
  4. Undo adding doorSensor
  5. A better alternative is to make a new component “SensorProxy” that abstracts away the knowledge of a variety of sensor types for the AlarmSystem.dm to improve the locality of change and separation of concerns.

The Alarm System Exercise © 2013 Verum 29

In this exercise you learn how you can use abstraction to improve the locality of change and separation of concerns in a design.

Learning targets:

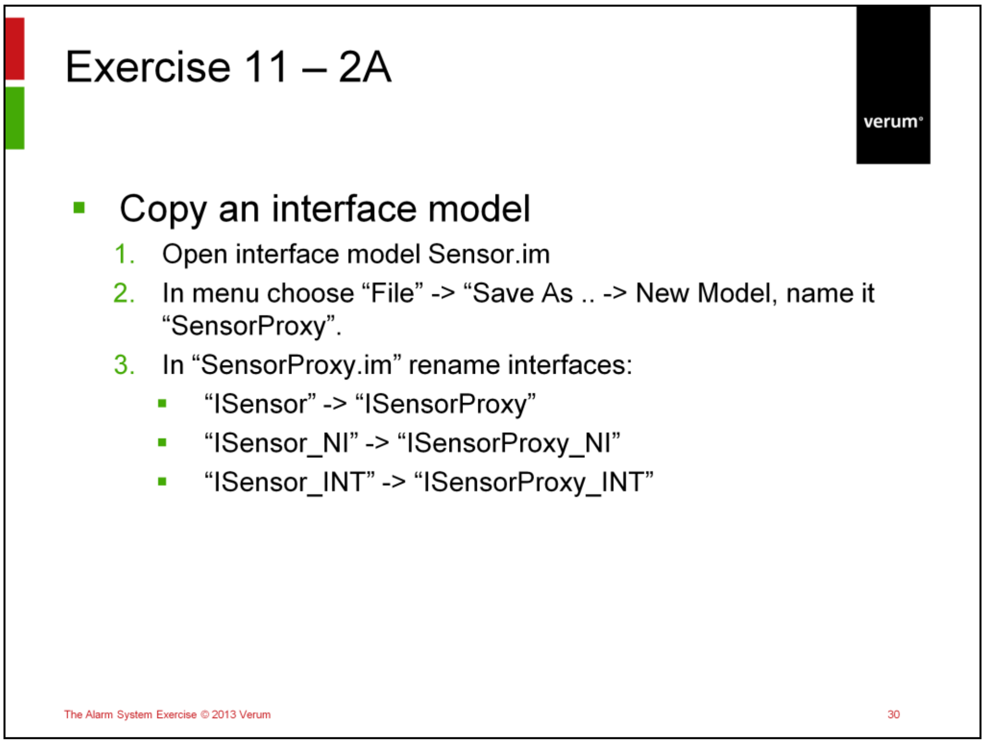
- How to use used service references

References course material:

- -

References user manual(s):

- ASD:Suite User Manual



Exercise 11 – 2A

verum®

- Copy an interface model
  1. Open interface model Sensor.im
  2. In menu choose “File” -> “Save As .. -> New Model, name it “SensorProxy”.
  3. In “SensorProxy.im” rename interfaces:
     - “ISensor” -> “ISensorProxy”
     - “ISensor_NI” -> “ISensorProxy_NI”
     - “ISensor_INT” -> “ISensorProxy_INT”

The Alarm System Exercise © 2013 Verum 30

In this exercise you learn how you can make a copy of a interface model

Learning targets:

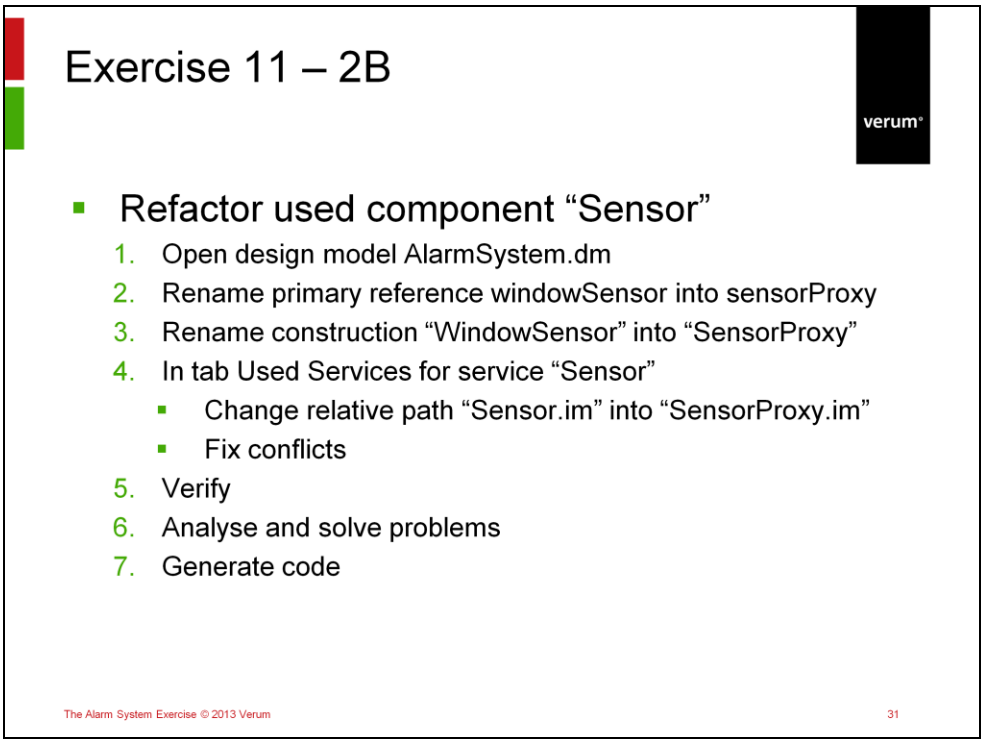
- How to use used service references

References course material:

- -

References user manual(s):

• ASD:Suite User Manual



Exercise 11 – 2B

verum®

- Refactor used component "Sensor"
  1. Open design model AlarmSystem.dm
  2. Rename primary reference windowSensor into sensorProxy
  3. Rename construction "WindowSensor" into "SensorProxy"
  4. In tab Used Services for service "Sensor"
     - Change relative path "Sensor.im" into "SensorProxy.im"
     - Fix conflicts
  5. Verify
  6. Analyse and solve problems
  7. Generate code

The Alarm System Exercise © 2013 Verum 31

In this exercise you learn how you can use another service / component.

Learning targets:

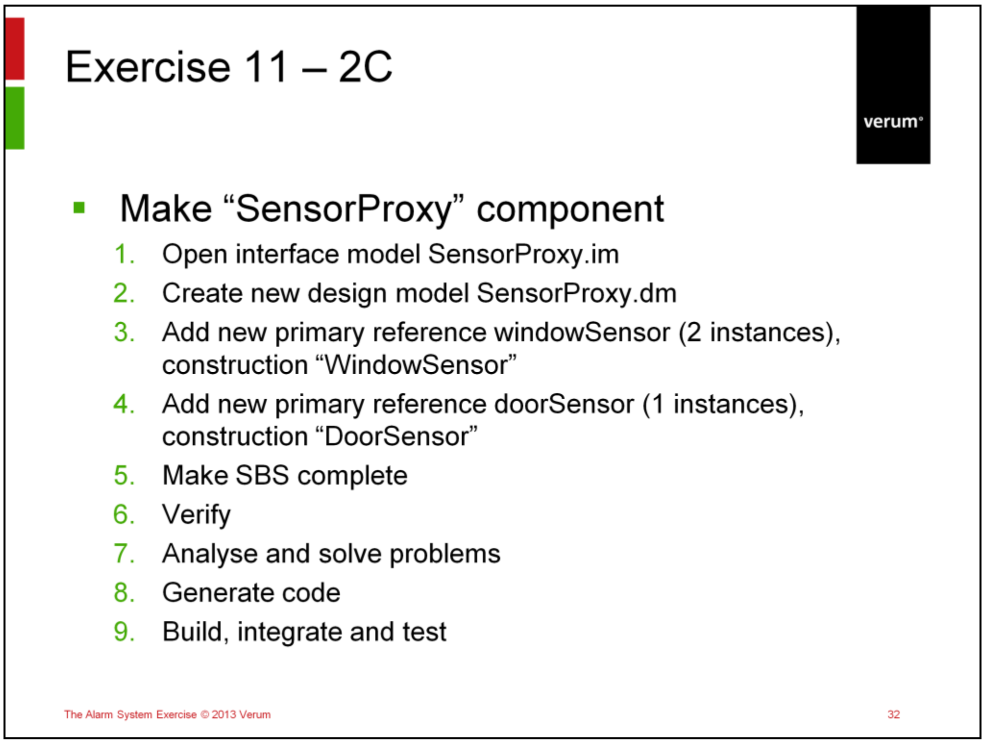
- How to use used service references

References course material:

- -

References user manual(s):

• ASD:Suite User Manual



Exercise 11 – 2C

verum

- Make "SensorProxy" component
  1. Open interface model SensorProxy.im
  2. Create new design model SensorProxy.dm
  3. Add new primary reference windowSensor (2 instances), construction "WindowSensor"
  4. Add new primary reference doorSensor (1 instances), construction "DoorSensor"
  5. Make SBS complete
  6. Verify
  7. Analyse and solve problems
  8. Generate code
  9. Build, integrate and test

The Alarm System Exercise © 2013 Verum 32

In this exercise you learn how you can create a set of identical implementations of the same interface and how you can use Used Service References to address a set of components with the same interface.

Learning targets:

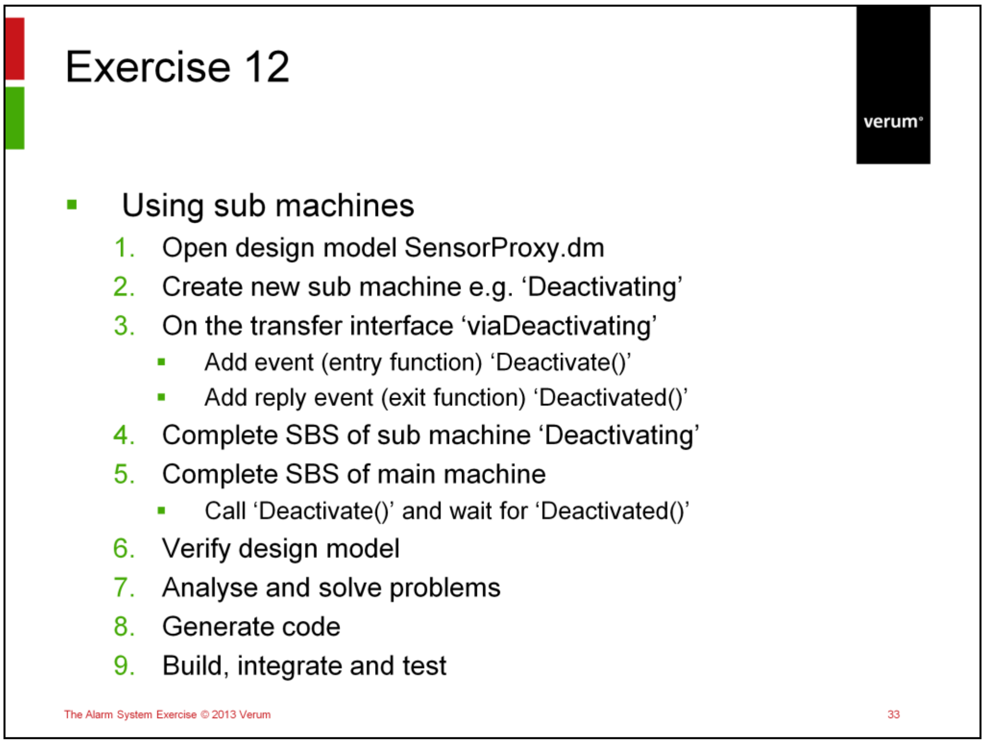
- How to use used service references
- How to use used service reference state variables and "that" operator

References course material:

- -

References user manual(s):

- ASD:Suite User Manual


Exercise 12

- Using sub machines
  1. Open design model SensorProxy.dm
  2. Create new sub machine e.g. ‘Deactivating’
  3. On the transfer interface ‘viaDeactivating’
     - Add event (entry function) ‘Deactivate()’
     - Add reply event (exit function) ‘Deactivated()’
  4. Complete SBS of sub machine ‘Deactivating’
  5. Complete SBS of main machine
     - Call ‘Deactivate()’ and wait for ‘Deactivated()’
  6. Verify design model
  7. Analyse and solve problems
  8. Generate code
  9. Build, integrate and test

The Alarm System Exercise © 2013 Verum 33

In this exercise you learn how you can use sub machines.

You can use sub machines for a repeating piece of state behaviour or for product life cycle phase like Initialisation, Termination, Starting, Stopping and so on.

The purpose of the sub machine in this exercise is to take care of all deactivation functionality in the SensorProxy. Because the SensorProxy only has to deactivate a set of sensors a separate sub machine may seem a bit overdone but in general it is a good approach; a Deactivation sub machine serves as an abstraction for the internal process of deactivation. On main machine level you only start the deactivation and wait for the result in a so-called super state. In the sub machine all details for deactivation are handled; any other activity is Illegal.

Note that state variables in main and sub machines are not shared.

Learning targets:

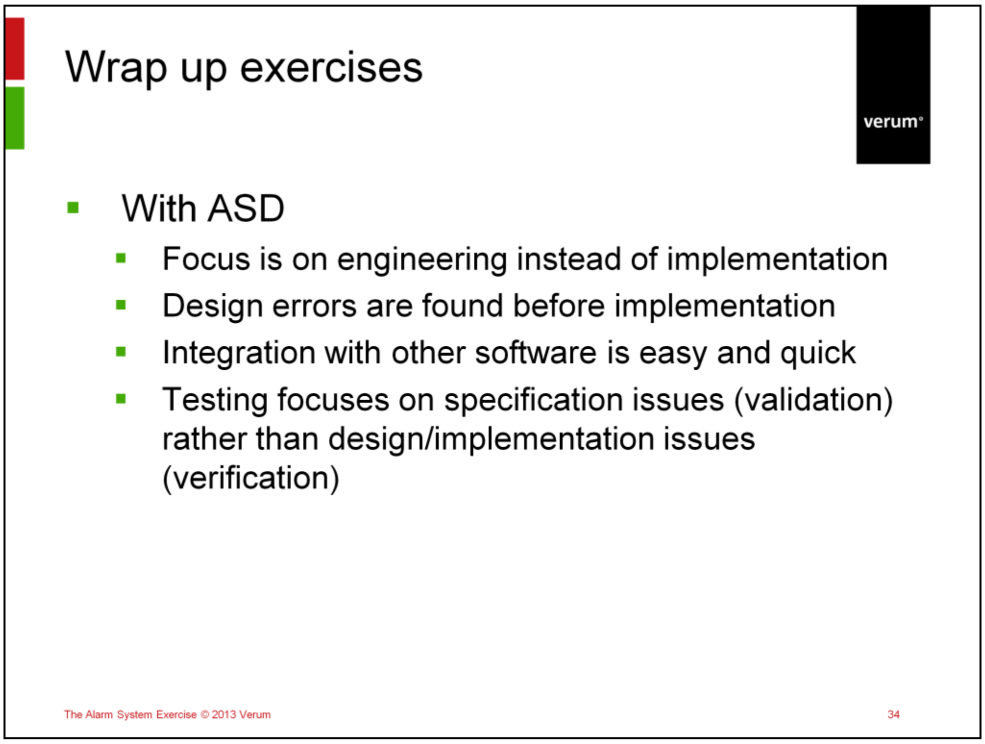
- How to use sub machines

References course material:

- Sequence Based Specifications – Designs with Sub Machines

References user manual(s):

• ASD:Suite User Manual



Wrap up exercises

verum

- With ASD
  - Focus is on engineering instead of implementation
  - Design errors are found before implementation
  - Integration with other software is easy and quick
  - Testing focuses on specification issues (validation) rather than design/implementation issues (verification)

The Alarm System Exercise © 2013 Verum 34

The hands-on experience while doing these exercises should have given you insights on the world behind ASD and also on how ASD impacts your work.